

Solutions to Time Variant Problems of Real-Time Expert Systems*

Show-Way Yeh and Chuan-lin Wu
Department of Electrical and Computer Engineering
The University of Texas
Austin, TX 78712

Chaw-Kwei Hung
Information Division
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, Ca 91109

ABSTRACT: Real-time expert systems for monitoring and control are driven by input data which changes with time. One of the subtle problems of this field is the propagation of time variant problems from rule to rule. This propagation problem is even complicated under a multiprogramming environment where the expert system may issue test commands to the system to get data and to access time consuming devices to retrieve data for concurrent reasoning. There are two approaches which have been used to handle the flood of input data. Snapshots can be taken to freeze the system from time to time. The expert system treats the system as a stationary one and traces changes by comparing consecutive snapshots. In the other approach, when an input is available, the rules associated with it are evaluated. For both approaches, if the premise condition of a fired rule is changed to being false, the downstream rules should be deactivated. If the status change is due to disappearance of a transient problem, actions taken by the fired downstream rules which are no longer true may need to be undone. If a downstream rule is being evaluated, it should not be fired. Three mechanisms for solving this problem are discussed in this paper: forward tracing, backward checking, and censor setting. In the forward tracing mechanism, when the premise conditions of a fired rule become false, the premise conditions of downstream rules which have been fired or are being evaluated due to the firing of that rule are reevaluated. A tree with its root at the rule being deactivated is traversed. In the backward checking mechanism, when a rule is being fired, the expert system checks back on the premise conditions of the upstream rules that result in evaluation of the rule to see whether it should be fired. The root of the tree being traversed is the rule being fired. In the censor setting mechanism, when a rule is to be evaluated, a censor is constructed based on the premise conditions of the upstream rules and the censor is evaluated just before the rule is fired. Unlike the backward checking mechanism, this one doesn't search the upstream rules. This paper explores the details of implementation of the three mechanisms.

1. Introduction

A real-time expert system is data-driven and is different from the conventional expert systems. For a conventional expert system, the premise conditions of rules do not change with time during the entire transaction, and the reasoning is directed by the operator. Establishing proper communication protocols between the expert system and the sensors and processors in the underlying system enables the expert system to monitor the

* This work is supported by Jet Propulsion Laboratory under contract GK857812.

underlying system and issue commands to control the underlying system directly. For a real-time expert system, the reasoning is triggered by the input data which are collected from the underlying system it controls [Cot87, Laf88, Lei87, Moo86, Pat85, Sau83]. The premise conditions of rules are predicates on the input data and on conclusions of other rules. Since the input data vary with time, the premise conditions of the real-time expert system rules vary accordingly. If a premise condition of a rule includes a predicate on the conclusion of another rule, then, if the premise condition of the latter is changed, the premise condition of the former will be affected because the conclusion of the latter is changed. That is, if the firing of a rule depends on another rule, then the changing of the premise condition of the latter will propagate to the former. This time variant propagation will continue if the conclusion of the former is a premise condition of other rules.

The input data come to the real-time expert system from processors or sensors distributed in the underlying system. Since the data vary with time, the real-time expert system also needs to reason in a time variant manner. The real-time expert system may take snapshots of the underlying system periodically. That is, it freezes the underlying system from time to time. Changes are traced by pairwise comparison of consecutive snapshots. The real-time expert system reasons using the input data from each snapshot and the changes between every two consecutive snapshots. If any changes are detected, the real-time expert system starts to investigate whether there are any problems. There is another mechanism in which, when an input data item is available, the system immediately investigates what happened and, if something has changed, takes actions if necessary.

This paper discusses problems of the time variant property of real-time expert systems. Although the underlying system addressed in this paper is assumed to be a real-time network system and all examples are based on network management, the problems are common to real-time monitor and control expert systems in all the various other application areas. The reader can easily convert the analogous problems to those of other real-time monitor and control expert systems. Section 2 describes the time variant problems of the real-time expert systems in detail and defines some of the terms we are using. Section 3 presents the reasons why a multiprogramming environment is necessary for some real-time systems. In a multiprogramming environment, a concurrency control mechanism is necessary to fire and to stop the firing of rules. Three mechanisms of deactivating rules being evaluated, forward tracing, backward checking, and dynamically censor setting, are discussed in section 4, 5, and 6. A comparison of these three mechanisms are presented in section 7. Finally, a brief conclusion is given in section 8.

2. Time Variant Problems

If any premise condition of a rule is a part of the conclusion of another rule, then the former is called a downstream rule of the latter and the latter is called an upstream rule of the former. If X is a downstream rule of Y and Y is a downstream rule of Z then X is called a downstream rule of Z and Z is called an upstream rule of X .

When the premise conditions of a fired rule become false, all downstream rules which have been fired or are being evaluated on the basis of the firing of that rule need to be reevaluated. For example, suppose that we have the following four rules:

if A then B and F
if B and C then D
if D then E
if F or G then H

Suppose that A becomes true at time T for a short period of time and C is true after time T as well. Then the above four rules should be fired sequentially after T . Suppose that by the time the rule *if B and C then D* is being evaluated A becomes false. Then all four of these rules should not be fired. For the already fired rule, *if A then B and F* , since actions of B and F have been done, complementary actions may need to be performed. For the rule being evaluated, *if B and C then D* , evaluation should be stopped and the rule should not be fired. The rule *if D then E* doesn't need to be evaluated.

If delay is tolerable, a transient time interval may be defined for each rule, and a rule is to be fired when the premise conditions remain true longer than the predefined interval [Lei86]. However, if there is not enough time available to make sure that the problem persists, actions should be taken place immediately and complementary actions should be taken if the problem is disappears. For example, if the real-time monitor and control expert system of a network receives a report saying that a command packet keeps being refused by another node and the packet must be received by the destination spacecraft in a very short time, the expert system may send commands to the sender to change the routing table to send the packet along the secondary path to the destination to meet the time limit constraint. However, the reason for refusal to receive packets may be that the input buffers of the receiver are temporarily full because too many nodes are sending packets to it at the same time. After the expert system sends the changing command to the sender, it may receive a report saying that the problem is gone. So, the expert system must send another command to the sender to change the routing table back to the original one to keep the traffic balanced.

Also, when an error has been corrected, the fired rules which managed the error should be deactivated. For example, if a node is dead, the routing tables of other nodes need to be changed to bypass the error. Once the node has been fixed, the routing tables need to be changed back to the original ones.

For all these cases, rules which have been fired need to be deactivated and evaluation of rules needs to be stopped when the premise conditions of upstream rules which resulted in firing or evaluation of the rules are changed to being false.

3. Execution Environment

Since the input data change with time, the actions to be taken will vary accordingly from one time to another. To maintain the whole system, changes in input data and actions taken in the past may need to be recorded somewhere in the system so that the expert system may trace what happened to the underlying system in the past. For example, suppose that each node in a network records the inbound and outbound packet headers in the network management database. Suppose that the expert system is reported that a packet is lost. Let C of the rule *if B and C then D* be the predicate of evaluation of recorded headers in the file storage of remote nodes. When the expert system evaluates C , it will send commands to all nodes along the path of the lost packet to retrieve the history headers to determine where is the problem. Before the data returns, the evaluating process can do nothing but waiting.

Meanwhile, the real-time expert system may need some current data from the underlying system to make decision. It may issue commands to processors or sensors in the underlying system to perform certain tests. For example, the rule *if A then B and F* is to determine that a node has generated noise. Suppose it is fired. Furthermore, let C of the rule *if B and C then D* be the predicate to evaluate the test value of the interface card of the node. The expert system will issue a command to the troublesome node to perform

the test. The expert system will make decision based on the test result.

Retrieving history data from remote data bases and instructing certain processors or sensors to perform tests usually are time consuming. When the expert system needs to do it, it cannot continue processing anything. On the other hand, while the expert system is waiting for the remote data or test results, the expert system cannot stop to wait because the underlying system status is changing and the flood of input data continues to arrive.

To process the new data and to manage new problems while the expert system is waiting, new processes need to be created. Therefore, expert systems of this kind must be implemented in a multiprogramming environment. A process is suspended when it needs to wait for information which is not currently available in the local system. It is resumed when the information for which it is waiting becomes available.

The time variant problems are even worse in the multiprogramming environment. For example, while a process is evaluating the rule *if B and C then D*, the input data associated with A may be changed by another process such that A becomes false. The latter, after deactivating the rule *if A then B and F* will find that, since B has been changed to be false, the rule *if B and C then D* needs to be reevaluated. So, two processes may not know each other and work independently and concurrently. They may evaluate the same rule and get different conclusions. Finally, conflict actions may be taken.

For convenience, we define the following terms. A process which evaluates the premise condition of a rule which is not fired and fires that rule if the premise condition becomes true is called an activating process. When input data are changed such that the premise condition of a rule becomes true, the rule will be fired and the conclusion will be committed to be true. If the commitment makes the premise condition of another downstream rule to be true, an activating process will fire the downstream rule, too. So, the activating process will propagate to the downstream rules until no downstream rules can be fired. On the other hand, A process which evaluates the premise condition of a rule which has been fired and deactivates that rule if the premise condition becomes false is called a deactivating process. When input data are changed such that the premise condition of a fired rule becomes false, the rule will be deactivated, complementary actions will be taken, and the conclusion will be committed to be false. If the commitment makes the premise condition of another fired downstream rule to be false, a deactivating process will deactivate the downstream rule, too. So, the deactivating process will propagate to the downstream rules until no fired downstream rules need to be fired.

If the rule *if F or G then H* is not fired and two activating processes evaluate F and G concurrently and respectively, then both processes will find that the premise condition becomes true and the rule will be fired twice. If the actions are expensive, for example, if H is to issue commands to some processors to perform certain tests, we want the actions to be taken only once. This problem can be easily solved by write-locking the firing status of each rule when an activating process checks to see whether the rule is fired or not. Similarly, if the rule *if B and C then D* has been fired and two deactivating processes evaluate B and C concurrently and respectively, then both processes will find that the premise condition becomes false and the rule will be deactivated twice. If the complementary actions are expensive, we want that the complementary actions are taken only once. This problem can be easily solved by write-locking the firing status of each rule, too.

Comparing a deactivating process with an activating process, we can find they are similar. When a deactivating process evaluates the premise condition of a fired rule, if the complement of the premise condition is true, it deactivates the rule and takes complementary actions. To deactivate a rule is just like to "fire" a rule to take the complementary actions. For example, the complement rule of the rule *if A then B and F* is *if A(past) and not A(now) then (not (B and F))* or *if (B and F) and not A then (not (B and F))*. To deactivate the first rule is the same as to activate either the second rule or the third rule.

However, since the expert system is to monitor and control the underlying system, the rules are to handle problems. If an activating process fires rules, most likely, the associated actions will lead the underlying to abnormal state to avoid the problems. In contrast with the activating process, when a deactivating process deactivates a fired rule, it usually means that the problems managed by the fired rule are gone. The associated complementary actions are to lead the underlying system back to normal state. Usually, the running cost of the underlying system in a normal state is lower than that in an abnormal state. Therefore, if the problems are transient, we prefer that the deactivating processes have higher priority to "catch up" the associated "front running" activating processes to stop them to keep the system in the normal states as much as possible. That is, we want that when a rule is fired, it is consistent with the current status of the input data. Three mechanisms, forward tracing, backward checking, and dynamic censor setting, of doing so are discussed.

4. Forward Tracing

In forward tracing, we just let the deactivating processes have higher priority than the activating processes. When the premise conditions of a fired rule become false, the premise conditions of the downstream rules which have been fired or are being evaluated due to the firing of that rule are reevaluated. The fired downstream rules are deactivated and the downstream rules being evaluated are stopped if the premise conditions become to be false. The deactivating process will traverse a tree rooted at the rule whose premise condition has just been changed to being false. Each leaf node is the first downstream rule which has not been fired or has been fired by other rules.

To deactivate a fired rule, the complementary actions are dependent on the actions of the rule. For example, if the action is to use alternative routing table, the complementary action is to use the principal routing table; if the action is to double the time-out period, the complementary action is to use the original time-out. There are not systematic ways to develop the complementary actions. They need to be developed rule by rule.

For the rules used as examples in the previous sections, when the rule *if B and C then D* is evaluated, *B* may be read into a working buffer. Then, when *A* becomes false, since the deactivating process has higher priority, *B* may be changed to being false before the activating process finishes evaluating the rule. The activating process acting on this rule may not know that *B* has been changed to being false and continues to evaluate the rule. Consequently, this rule and the rule *if D then E* will be fired.

It is apparent that concurrency control mechanisms are needed to make sure that changes of input data can affect the evaluation of the rules being evaluated. The traditional locking mechanisms may not be adequate to solve this problem because the deactivating process is designed to update data which have been read by the activating process. If the traditional locking mechanisms are applied, the activating process will read-lock the items it reads. This prevents the deactivating process from updating the items they have been locked. The deactivating process can never "catch up" with the "front

running" activating processes. Since the deactivating process has higher priority than the activating process, the traditional locking mechanisms are not applicable for the forward tracing mechanism.

Soft lock [Ege86, Yeh87] is a locking mechanism for software design data base systems. Since a designer may spend a long time figuring out how to design or update an item which can be a requirement, a specification, a circuit design, or a module of code, for each item there may be several versions. The designer may wish to reference others' work while he is doing his own. If the traditional locking mechanisms are applied to these systems, no one can update an item which is being referenced by others and no one can reference an item which is being updated by others. This will block designers from continuing with their work. Soft lock is designed to set a conflict mark on an item when a conflict between two transactions is detected. The two designers will be informed either as soon as the conflict is detected or at the end of the transaction that ends sooner. Then, the designers will resolve the conflict.

Similarly, we may allow the deactivating process to break a soft lock set by the activating process. When the activating process starts to evaluate a rule, it softly locks the premise conditions and then reads them. When the deactivating process wants to change any premise condition of a rule, it checks first whether a soft read lock has been set on the premise condition. If yes, it breaks the lock and updates the value. Finally, when the activating process finishes evaluating the rule, it checks to see if any soft locks have been broken. If yes, it reevaluates the rule. Otherwise, it fires the rule and unlocks the locks.

For the OR rule *if F or G then H*, when *A* is true, *F* is true. No matter whether *G* is true or false, this rule will be fired. But, *G* may become true at any time. If, when *A* becomes false, *G* has become true, this rule will not be deactivated and is a leaf node of the tree which is being traversed by the deactivating process.

Obviously, there is no guarantee that all deactivating processes can "catch up" with the activating processes.

5. Backward Checking

To overcome the drawback of forward tracing that the deactivating process may not be able to "catch up" with the associated activating processes acting on downstream rules before the rules are fired, the expert system may check back on the premise conditions of all upstream rules to see whether any of them have been changed right before the rule being evaluated is fired. Backward checking means that when a rule is being fired, the expert system checks the premise conditions of the upstream rules which result in evaluation of the rule to see whether they are still true.

For the rules used as examples in the previous sections, when the rule *if B and C then D* is finished being evaluated, the expert system checks back to see whether *B* and *C* are still true. Since *B* is produced from *A* being true, the expert system needs to check whether *A* and *C* are still true. That is, a tree rooted at the rule being evaluated is traversed. Similarly, when the rule *if D then E* is finished being evaluated, the expert system checks back to see whether *D* is still true. Since *D* is produced from *B* and *C* being true and *B* is produced from *A* being true, the expert system needs to check back to see whether *A* and *C* are still true.

For the rule *if F or G then H* if only *F* or *G* changes to being false and the other remains true, the rule cannot be deactivated. Since *F* is produced by the rule *if A then B and F*, the backward checking mechanism needs to check the premise conditions of both rules. Only if both *A* and *G* become false should the rule be deactivated. If only one of *F* and *G* is true and the other is false or unknown when the rule is evaluated, then the checking back mechanism only checks the upstream rules of the predicate whose truth value is true.

The rule is fired only if its premise conditions are still true. If there is not enough time to check back, the rule is fired without checking back. Since the forward deactivating process and the backward checking process head toward each other, they will meet each other somewhere in the middle. This guarantees that if a premise condition is changed, all downstream rules which are being evaluated will not be fired if the change will deactivate them.

The locking mechanism needed for the backward checking mechanism is similar to that for the forward tracing mechanism. The backward checking activating process will softly lock the premise conditions of an upstream rule before it starts to evaluate the premise conditions. The deactivating process can break soft locks to update the premise conditions. The locks on the rules which have been fired can be traditional locks or soft locks.

6. Dynamically Setting Censors

In the backward checking mechanism, when a rule is evaluated, all premise conditions of the upstream rules need to be checked. A drawback of this mechanism is that the upstream rules need to be searched and searching may be expensive.

Variable precision logic rules have the form *if X then Y unless Z*. If, when *X* is true, *Y* is true with a high probability *x* and is false only if *Z* is true with low probability $1-x$, then, if there is not enough time to evaluate *Z*, we may fire the rule *if X then Y*. The probability of getting the correct result is high. *Z* is called a censor [Mic86, Had86].

Dynamically setting censors means that, when a rule is to be evaluated, a censor is constructed based on the premise conditions of the upstream rules and the censor is evaluated just before the rule is fired. For the rules in the previous sections, suppose *A* and *C* are predicates on input data and are true. The rule *if A then B and F* will be evaluated and be fired. When it is being evaluated, the rule is reconstructed as *if A then B and F unless (not A)*. This rule will not be fired if *A* becomes false while the rule is being evaluated. If *A* continues to be true and this rule is fired, since *B* becomes true, the rule *if B and C then D* should be fired. Therefore, since what may be changed are *A* and *C*, before we evaluate the rule, the rule is reconstructed to be *if B and C then D unless (not A or not C)*. Then, if *A* or *C* changes to being false while this rule is being evaluated, this rule will not be fired. Similarly, when we are to evaluate the rule *if D then E* is to be evaluated, since what the activating process needs to check is *A* and *C*, we reconstruct the rule to be *if D then E unless (not A or not C)*. It will not be fired if *A* or *C* becomes false while it is being evaluated.

The censor looks awful if the premise condition of the rule is AND of predicates because all of the predicates are taken into the censor. If the premise condition of a rule is inclusive OR of predicates, then the censor is the complements of the predicate which makes the rule to be fired. This is because that the censor is to prevent from transient

problems. When the rule is about to be fired, what we worry about is whether the premise conditions of the upstream rules have been changed such that the current rule should not be fired. For example, if F is changed to being true and the truth value of G is unknown, the rule *if F or G then H* should be evaluated and to be fired. But, before evaluating it, it is reconstructed to be *if F or G then H unless (not A)* where the censor is inherited from the rule *if A then B and F by F* . So, when the activating process finds that this rule should be fired, it checks to see whether A is still true. If yes, fires the rule. Otherwise, it stops itself.

This mechanism only evaluates predicates on input data. The upstream rules are not evaluated and are not searched. When evaluating a rule results from firing other rules, it is easy to construct the censor. It is the combination of the censors of the upstream rules and the input data of the current rule. For example, the premise of the first rule, A , is a predicate on the input data. Its censor is the complement of the predicate. For the second rule, B is a predicate which is the conclusion of the first rule. The censor of the first rule is inherited. C is a predicate on the input data. So, the censor is the inclusive OR of the inherited predicate from the first rule and the complement of C . For the third rule, the premise predicate is the conclusion of the second rule. So, the censor of the second rule is inherited to be the censor of the third rule. For the fourth rule, since the premise condition is the inclusive OR of two predicates, the censor is the complement of the one which makes the rule be evaluated. If the predicate is conclusion of an upstream rule, the censor is inherited from the upstream rule.

Some of the problems can never be transient problems. For example, if a packet is reported to be lost by the operator, this problem cannot be gone in a short time. Another kind of problem, for example, error rate of a link is detected to be a little bit too high 10 times in an hour. If it is detected only once, that may be tolerable. But, the problem is detected by counting how many times it happened. Once the number is higher than the threshold, the problem is identified. To count the number of times, once the high error rate is detected, it will counted once and not be changed. For this kind of problem, since no transient problem, censor is not necessary. That is, censor is set only when the input data do be able to be changed transiently.

7. Comparison of the Three Mechanisms

The simplest one among the three mechanisms is the forward tracing mechanism. Since it doesn't guarantee that the activating processes can be caught up by the associated deactivating processes when input data of upstream rules are changed to being true transiently and then are changed to being false, if the cost of firing the rule is expensive, this mechanism should not be used. The other two mechanisms guarantee that if the input data of upstream rules are changed to make the rules false while the rule is being evaluated, the activating process will be stopped. The backward checking mechanism needs to search the upstream rules to reevaluate the predicates. It not only reevaluates the input data associated with the rule being evaluated, but also reevaluates the intermediate predicates. While the dynamic setting censor mechanism only reevaluates the input data and doesn't need to search for the intermediate upstream rules. So, the backward checking is less efficient than the dynamic setting censor mechanism. If the actions of a rule are very expensive and the rule needs to be evaluated in very short time, dynamic setting censor mechanism is the best choice.

8. Conclusion

In this paper, we address some characteristics of real-time monitor and control expert systems. The input data are collected from the underlying system and are changing with time. When the input data change, the rules which have been fired or are being fired on the basis of the changed data need to be reevaluated. When the premise condition of a rule is changed, the downstream rules need to be reevaluated, too. That is, the time dependent changes may propagate from rule to rule. If a fired rule needs to be deactivated, complementary actions need to be taken to overcome the change. If the premise condition of a rule that is being evaluated is changed, the activating process needs to be stopped. This time variant problem gets worse in the multiprogramming environment. Three mechanisms are discussed in this paper. In the forward tracing mechanism, the deactivating process has higher priority than the activating process. This mechanism cannot guarantee that the deactivating process can catch up with the activating process. To overcome this drawback, a backward checking mechanism may be used that, when a rule is about to be fired, checks the upstream rules to see whether the rule still should be fired. The drawback of this mechanism is that extra searching is needed and is expensive. In the censor setting mechanism, changes of input data are inherited from rule to rule to eliminate the searching overhead of the backward checking mechanism. Since the backward checking and the dynamically setting censor mechanisms need some redundant work, the rules need to be investigated one by one to determine whether it has transient problem. If not, these two mechanisms should not be applied.

Reference

- [Cot87] Cotthem, H., Mathonet, R., and Vanryckeghem, L., "Dantes--An Expert System Shell Dedicated to Real-Time Network Troubleshooting", Workshop on Expert Systems and Network Operation, IEEE, ICC, 1987.
- [Dvo87] Dvorak, D., "Expert Systems for Monitoring and Control--A Survey", University of Texas at Austin Technical Report AI87-55, May 1987.
- [Ege86] Ege, A., Ellis C., and Wexelblat, A., "Design and Implementation of GORDION, An Object Base Management System", MCC Tec. Rep. No. STP-139-86(Q), April 1986.
- [Fer86] Fertig, K., Andrews, A., and Wang, C., "Knowledge-Based Management and Control of Communications Networks", Milcom'86, 28.6.
- [Goy87] Goyal, S., and Kopeikina, L., "Evolution of Intelligent Telecommunication Networks", Workshop on Expert Systems and Network Operation, IEEE, ICC, 1987.
- [Had86] Haddawy, P., "Implementation of and Experiments with a Variable Precision Logic Inference System", AAAI, 1986.
- [JPL86] "Technical Requirements for the GCF Upgrade Task", JPL, 4800 Oak Grove Dr., Pasadena, CA 91109, 1986.
- [Kat87] Katsuyama, Y., "Expert System for Digital Transmission Network Troubleshooting", Workshop on Expert Systems and Network Operation, IEEE, ICC, 1987.
- [Laf88] Laffey, T., Cox, P., Schmidt, J., Kao, S., and Read, J., "Real-Time Knowledge-Based Systems", AI Magazine, Spring 1988, pp 28-45.
- [Lei87] Leinweber, D., "Real-Time Expert Systems for Space Applications", AFCEA Symposium "Space Technological Challenges for the Future", US Naval Academy, 1987.
- [Mic86] Michalski, R, and Winston, P., "Variable Precision Logic", Artificial Intelligence, Vol 29, No 2, 1986.

- [Moo86] Moore, R., "Expert Systems in On-Line Process Control", Proc. CPC-3 Conf., 1986.
- [Pat85] Paterson, A., Sachs, P., and Turner, M., "ESCORT: the Application of Causal Knowledge to Real-Time Process Control", Expert Systems 85, Cambridge University Press 1985.
- [Red86] Reddy, Y., and Uppuluri, S., "Intelligent Systems Technology in Network Operations Management", IEEE ICC, 1986, 39.1.
- [Sau83] Sauers, R., and Walsh, R., "On the Requirements of Future Expert Systems", Proc. 8th IJCAI, 1983.
- [Ter87] Terplan, K., "Communication Network Management", Prentice-Hall, Inc., 1987.
- [Yeh87] Yeh, S., Ellis, C., Ege, A., and Korth, H., "Mean Value Performance Analysis of Two Locking Mechanisms in Centralized Design Environment", International Journal of Information Science, 1988.